SSJ User's Guide

Package `util`

General basic utilities

Version: December 17, 2014

This document describes a set of basic utilities used in the Java software developed in the *simulation laboratory* of the DIRO, at the Université de Montréal. Many of these tools were originally implemented in the Modula-2 language and have been translated in C and then in Java, with some adaptations along the road.

Contents

Num 2
TextDataReader 7
PrintfFormat 10
TableFormat 15
AbstractChrono 16
SystemTimeChrono 18
GlobalCPUTimeChrono 19
ThreadCPUTimeChrono 20
Chrono 21
ChronoSingleThread 22
TimeUnit 23
Systeme 25
ArithmeticMod 26
BitVector 28
BitMatrix 31
DMatrix 34
RootFinder 37
MultivariateFunction 38
RatioFunction 39
Misc 40
JDBCManager 42
ClassFinder 47
NameConflictException 49
Introspection 50
TransformingList 52
DoubleArrayComparator 53

Overview of package `util.io` **54**
DataWriter 56
AbstractDataWriter 58
CachedDataWriter 59
TextDataWriter 61
BinaryDataWriter 63
DataReader 66
AbstractDataReader 68
BinaryDataReader 70
DataField 71

Num

This class provides a few constants and some methods to compute numerical quantities such as factorials, combinations, gamma functions, and so on.

```
package umontreal.iro.lecuyer.util;

public class Num
```

Constants

`public static final double DBL_EPSILON = 2.2204460492503131e-16;`

Difference between 1.0 and the smallest `double` greater than 1.0.

`public static final int DBL_MAX_EXP = 1024;`

Largest `int` x such that 2^{x-1} is representable (approximately) as a `double`.

`public static final int DBL_MIN_EXP = -1021;`

Smallest `int` x such that 2^{x-1} is representable (approximately) as a normalised `double`.

`public static final int DBL_MAX_10_EXP = 308;`

Largest `int` x such that 10^x is representable (approximately) as a `double`.

`public static final double DBL_MIN = 2.2250738585072014e-308;`

Smallest normalized positive floating-point `double`.

`public static final double LN_DBL_MIN = -708.3964185322641;`

Natural logarithm of `DBL_MIN`.

`public static final int DBL_DIG = 15;`

Number of decimal digits of precision in a `double`.

`public static final double EBASE = 2.7182818284590452354;`

The constant e.

`public static final double EULER = 0.57721566490153286;`

The Euler-Mascheroni constant.

`public static final double RAC2 = 1.41421356237309504880;`

The value of $\sqrt{2}$.

`public static final double IRAC2 = 0.70710678118654752440;`

The value of $1/\sqrt{2}$.

`public static final double LN2 = 0.69314718055994530941;`

The values of $\ln 2$.

`public static final double ILN2 = 1.44269504088896340737;`

The values of $1/\ln 2$.

`public static final double MAXINTDOUBLE = 9007199254740992.0;`

Largest integer $n_0 = 2^{53}$ such that any integer $n \leq n_0$ is represented exactly as a `double`.

`public static final double MAXTWOEXP = 64;`

Powers of 2 up to `MAXTWOEXP` are stored exactly in the array `TWOEXP`.

`public static final double TWOEXP[]`

Contains the precomputed positive powers of 2. One has `TWOEXP[j]`$= 2^j$, for $j = 0, \ldots, 64$.

`public static final double TEN_NEG_POW[]`

Contains the precomputed negative powers of 10. One has `TEN_NEG_POW[j]`$= 10^{-j}$, for $j = 0, \ldots, 16$.

Methods

`public static int gcd (int x, int y)`

Returns the greatest common divisor (gcd) of x and y.

`public static long gcd (long x, long y)`

Returns the greatest common divisor (gcd) of x and y.

`public static double combination (int n, int s)`

Returns the value of $\binom{n}{s}$, the number of different combinations of s objects amongst n.

`public static double lnCombination (int n, int s)`

Returns the natural logarithm of $\binom{n}{s}$, the number of different combinations of s objects amongst n.

`public static double factorial (int n)`

Returns the value of $n!$

`public static double lnFactorial (int n)`

Returns the value of $\ln(n!)$, the natural logarithm of factorial n. Gives 16 decimals of precision (relative error $< 0.5 \times 10^{-15}$).

`public static double lnFactorial (long n)`

Returns the value of $\ln(n!)$, the natural logarithm of factorial n. Gives 16 decimals of precision (relative error $< 0.5 \times 10^{-15}$).

`public static double factoPow (int n)`

Returns the value of $n!/n^n$.

`public static double[][] calcMatStirling (int m, int n)`

Computes and returns the Stirling numbers of the second kind

$$M[i,j] = \left\{ {j \atop i} \right\} \quad \text{for } 0 \le i \le m \text{ and } 0 \le i \le j \le n. \tag{1}$$

See [4, Section 1.2.6]. The matrix M is the transpose of Knuth's (1973).

`public static double log2 (double x)`

Returns $\log_2(\texttt{x})$.

`public static double lnGamma (double x)`

Returns the natural logarithm of the gamma function $\Gamma(x)$ evaluated at `x`. Gives 16 decimals of precision, but is implemented only for $x > 0$.

`public static double lnBeta (double lam, double nu)`

Computes the natural logarithm of the Beta function $B(\lambda, \nu)$. It is defined in terms of the Gamma function as

$$B(\lambda, \nu) = \frac{\Gamma(\lambda)\Gamma(\nu)}{\Gamma(\lambda + \nu)}$$

with `lam` $= \lambda$ and `nu` $= \nu$.

`public static double digamma (double x)`

Returns the value of the logarithmic derivative of the Gamma function $\psi(x) = \Gamma'(x)/\Gamma(x)$.

`public static double trigamma (double x)`

Returns the value of the trigamma function $d\psi(x)/dx$, the derivative of the digamma function, evaluated at x.

`public static double tetragamma (double x)`

Returns the value of the tetragamma function $d^2\psi(x)/d^2x$, the second derivative of the digamma function, evaluated at x.

`public static double gammaRatioHalf (double x)`

Returns the value of the ratio $\Gamma(x + 1/2)/\Gamma(x)$ of two gamma functions. This ratio is evaluated in a numerically stable way. Restriction: $x > 0$.

`public static double sumKahan (double[] A, int n)`

Implementation of the Kahan summation algorithm. Sums the first n elements of A and returns the sum. This algorithm is more precise than the naive algorithm. See `http://en.wikipedia.org/wiki/Kahan_summation_algorithm`.

`public static double harmonic (long n)`

Computes the n-th harmonic number $H_n = \sum_{j=1}^{n} 1/j$.

`public static double harmonic2 (long n)`

Computes the sum

$$\sum_{-n/2<j\le n/2}{}' \frac{1}{|j|},$$

where the symbol $\sum'$ means that the term with $j = 0$ is excluded from the sum.

`public static double volumeSphere (double p, int t)`

Returns the volume V of a sphere of radius 1 in t dimensions using the norm L_p. It is given by the formula

$$V = \frac{[2\Gamma(1+1/p)]^t}{\Gamma(1+t/p)}, \qquad p > 0,$$

where Γ is the gamma function. The case of the sup norm L_∞ is obtained by choosing $p = 0$. Restrictions: $p \ge 0$ and $t \ge 1$.

`public static double bernoulliPoly (int n, double x)`

Evaluates the Bernoulli polynomial $B_n(x)$ of degree n at x. Only degrees $n \le 8$ are programmed for now. The first Bernoulli polynomials of even degree are:

$$\begin{aligned} B_0(x) &= 1 \\ B_2(x) &= x^2 - x + 1/6 \\ B_4(x) &= x^4 - 2x^3 + x^2 - 1/30 \\ B_6(x) &= x^6 - 3x^5 + 5x^4/2 - x^2/2 + 1/42 \\ B_8(x) &= x^8 - 4x^7 + 14x^6/3 - 7x^4/3 + 2x^2/3 - 1/30. \end{aligned} \tag{2}$$

`public static double evalCheby (double a[], int n, double x)`

Evaluates a series of Chebyshev polynomials T_j at x over the basic interval $[-1,\ 1]$, using the method of Clenshaw [1], i.e., computes and returns

$$y = \frac{a_0}{2} + \sum_{j=1}^{n} a_j T_j(x).$$

`public static double evalChebyStar (double a[], int n, double x)`

Evaluates a series of shifted Chebyshev polynomials T_j^* at x over the basic interval $[0,\ 1]$, using the method of Clenshaw [1], i.e., computes and returns

$$y = \frac{a_0}{2} + \sum_{j=1}^{n} a_j T_j^*(x).$$

`public static double besselK025 (double x)`

Returns the value of $K_{1/4}(x)$, where K_ν is the modified Bessel's function of the second kind. The relative error on the returned value is less than 0.5×10^{-6} for $x > 10^{-300}$.

`public static double expBesselK1 (double x, double y)`

Returns the value of $e^x K_1(y)$, where K_1 is the modified Bessel function of the second kind of order 1. Restriction: $y > 0$.

`public static double erf (double x)`

Returns the value of `erf`(x), the error function. It is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x dt\, e^{-t^2}.$$

`public static double erfc (double x)`

Returns the value of `erfc`(x), the complementary error function. It is defined as

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty dt\, e^{-t^2}.$$

`public static double erfInv (double u)`

Returns the value of $\texttt{erf}^{-1}(u)$, the inverse of the error function. If $u =$ `erf`(x), then $x = \texttt{erf}^{-1}(u)$.

`public static double erfcInv (double u)`

Returns the value of $\texttt{erfc}^{-1}(u)$, the inverse of the complementary error function. If $u =$ `erfc`(x), then $x = \texttt{erfc}^{-1}(u)$.

TextDataReader

Provides static methods to read data from text files.

```
package umontreal.iro.lecuyer.util;

public class TextDataReader
```

`public static double[] readDoubleData (Reader input) throws IOException`

Reads an array of double-precision values from the reader `input`. For each line of text obtained from the given reader, this method trims whitespaces, and parses the remaining text as a double-precision value. This method ignores every character other than the digits, the plus and minus signs, the period (`.`), and the letters `e` and `E`. Moreover, lines starting with a pound sign (`#`) are considered as comments and thus skipped. The method returns an array containing all the parsed values.

`public static double[] readDoubleData (URL url) throws IOException`

Connects to the URL referred to by the URL object `url`, and calls `readDoubleData` to obtain an array of double-precision values from the resource.

`public static double[] readDoubleData (File file) throws IOException`

Opens the file referred to by the file object `file`, and calls `readDoubleData` to obtain an array of double-precision values from the file.

`public static double[] readDoubleData (String file) throws IOException`

Opens the file with name `file`, and calls `readDoubleData` to obtain an array of double-precision values from the file.

`public static int[] readIntData (Reader input) throws IOException`

This is equivalent to `readDoubleData`, for reading integers.

`public static int[] readIntData (URL url) throws IOException`

Connects to the URL referred to by the URL object `url`, and calls `readIntData` to obtain an array of integers from the resource.

`public static int[] readIntData (File file) throws IOException`

This is equivalent to `readDoubleData`, for reading integers.

`public static int[] readIntData (String file) throws IOException`

This is equivalent to `readDoubleData`, for reading integers.

`public static String[] readStringData (Reader input) throws IOException`

Reads an array of strings from the reader `input`. For each line of text obtained from the given reader, this method trims leading and trailing whitespaces, and stores the remaining string. Lines starting with a pound sign (`#`) are considered as comments and thus skipped. The method returns an array containing all the read strings.

`public static String[] readStringData (URL url) throws IOException`

Connects to the URL referred to by the URL object `url`, and calls `readStringData` to obtain an array of integers from the resource.

`public static String[] readStringData (File file) throws IOException`

This is equivalent to `readDoubleData`, for reading strings.

`public static String[] readStringData (String file) throws IOException`

This is equivalent to `readDoubleData`, for reading strings.

```
public static double[][] readDoubleData2D (Reader input)
                                            throws IOException
```

Uses the reader `input` to obtain a 2-dimensional array of double-precision values. For each line of text obtained from the given reader, this method trims whitespaces, and parses the remaining text as an array of double-precision values. Every character other than the digits, the plus (`+`) and minus (`-`) signs, the period (`.`), and the letters `e` and `E` are ignored and can be used to separate numbers on a line. Moreover, lines starting with a pound sign (`#`) are considered as comments and thus skipped. The lines containing only a semicolon sign (`;`) are considered as empty lines. The method returns a 2D array containing all the parsed values. The returned array is not always rectangular.

`public static double[][] readDoubleData2D (URL url) throws IOException`

Connects to the URL referred to by the URL object `url`, and calls `readDoubleData2D` to obtain a matrix of double-precision values from the resource.

`public static double[][] readDoubleData2D (File file) throws IOException`

Opens the file referred to by the file object `file`, and calls `readDoubleData2D` to obtain a matrix of double-precision values from the file.

```
public static double[][] readDoubleData2D (String file)
                                            throws IOException
```

Opens the file with name `file`, and calls `readDoubleData2D` to obtain a matrix of double-precision values from the file.

`public static int[][] readIntData2D (Reader input) throws IOException`

This is equivalent to `readDoubleData2D`, for reading integers.

`public static int[][] readIntData2D (URL url) throws IOException`

Connects to the URL referred to by the URL object `url`, and calls `readDoubleData` to obtain a matrix of integers from the resource.

`public static int[][] readIntData2D (File file) throws IOException`

This is equivalent to `readDoubleData2D`, for reading integers.

`public static int[][] readIntData2D (String file) throws IOException`

This is equivalent to `readDoubleData2D`, for reading integers.

```
public static String[][] readCSVData (Reader input, char colDelim,
                                      char stringDelim)
                                      throws IOException
```

Reads comma-separated values (CSV) from reader `input`, and returns a 2D array of strings corresponding to the read data. Lines are delimited using line separators `\r`, `\n`, and `\r\n`. Each line contains one or more values, separated by the column delimiter `colDelim`. If a string of characters is surrounded with the string delimiter `stringDelim`, any line separator and column separator appear in the string. The string delimiter can be inserted in such a string by putting it twice. Usually, the column delimiter is the comma, and the string delimiter is the quotation mark. The following example uses these default delimiters.

```
"One","Two","Three"
 1,2,3
"String with "" delimiter",n,m
```

This produces a matrix of strings with dimensions 3×3. The first row contains the strings `One`, `Two`, and `Three` while the second row contains the strings `1`, `2`, and `3`. The first column of the last row contains the string `String with " delimiter`.

```
public static String[][] readCSVData (URL url, char colDelim,
                                      char stringDelim)
                                      throws IOException
```

Connects to the URL referred to by the URL object `url`, and calls `readCSVData` to obtain a matrix of strings from the resource.

```
public static String[][] readCSVData (File file, char colDelim,
                                      char stringDelim)
                                      throws IOException
```

This is equivalent to `readDoubleData2D`, for reading strings.

```
public static String[][] readCSVData (String file, char colDelim,
                                      char stringDelim)
                                      throws IOException
```

This is equivalent to `readDoubleData2D`, for reading strings.

PrintfFormat

This class acts like a `StringBuffer` which defines new types of `append` methods. It defines certain functionalities of the ANSI C `printf` function that also can be accessed through static methods. The information given here is strongly inspired from the `man` page of the C `printf` function.

Most methods of this class format numbers for the English US locale only. One can use the Java class `Formatter` for performing locale-independent formatting.

```
package umontreal.iro.lecuyer.util;

public class PrintfFormat implements CharSequence, Appendable
```

Constants

`public static final String NEWLINE`

End-of-line symbol or line separator. It is "\n" for Unix/Linux, "\r\n" for MS-DOS/MS-Windows, and "\r" for Apple OSX.

`public static final String LINE_SEPARATOR`

End-of-line symbol or line separator. Same as `NEWLINE`.

Constructors

`public PrintfFormat()`

Constructs a new buffer object containing an empty string.

`public PrintfFormat (int length)`

Constructs a new buffer object with an initial capacity of `length`.

`public PrintfFormat (String str)`

Constructs a new buffer object containing the initial string `str`.

Methods

`public PrintfFormat append (String str)`

Appends `str` to the buffer.

`public PrintfFormat append (int fieldwidth, String str)`

Uses the `s` static method to append `str` to the buffer. A minimum of `fieldwidth` characters will be used.

`public PrintfFormat append (double x)`

Appends `x` to the buffer.

`public PrintfFormat append (int fieldwidth, double x)`
Uses the `f` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used.

`public PrintfFormat append (int fieldwidth, int precision, double x)`
Uses the `f` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used with the given `precision`.

`public PrintfFormat append (int x)`
Appends `x` to the buffer.

`public PrintfFormat append (int fieldwidth, int x)`
Uses the `d` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used.

`public PrintfFormat append (long x)`
Appends `x` to the buffer.

`public PrintfFormat append (int fieldwidth, long x)`
Uses the `d` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used.

```
public PrintfFormat append (int fieldwidth, int accuracy, int precision,
                            double x)
```
Uses the `format` static method with the same four arguments to append `x` to the buffer.

`public PrintfFormat append (char c)`
Appends a single character to the buffer.

`public void clear()`
Clears the contents of the buffer.

`public StringBuffer getBuffer()`
Returns the `StringBuffer` associated with that object.

`public String toString()`
Converts the buffer into a `String`.

`public static String s (String str)`
Same as `s (0, str)`. If the string `str` is null, it returns the string "null".

`public static String s (int fieldwidth, String str)`
Formats the string `str` like the `%s` in the C `printf` function. The `fieldwidth` argument gives the minimum length of the resulting string. If `str` is shorter than `fieldwidth`, it is left-padded with spaces. If `fieldwidth` is negative, the string is right-padded with spaces if necessary. The `String` will never be truncated. If `str` is null, it calls `s (fieldwidth, ``null'')`. The `fieldwidth` argument has the same effect for the other methods in this class.

Integers

`public static String d (long x)`

Same as `d (0, 1, x)`.

`public static String d (int fieldwidth, long x)`

Same as `d (fieldwidth, 1, x)`.

`public static String d (int fieldwidth, int precision, long x)`

Formats the long integer `x` into a string like `%d` in the C `printf` function. It converts its argument to decimal notation, `precision` gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. When zero is printed with an explicit precision 0, the output is empty.

`public static String format (long x)`

Same as `d (0, 1, x)`.

`public static String format (int fieldwidth, long x)`

Converts a long integer to a `String` with a minimum length of `fieldwidth`, the result is right-padded with spaces if necessary but it is not truncated. If only one argument is specified, a `fieldwidth` of 0 is assumed.

`public static String formatBase (int b, long x)`

Same as `formatBase (0, b, x)`.

`public static String formatBase (int fieldwidth, int b, long x)`

Converts the integer `x` to a `String` representation in base `b`. Restrictions: $2 \leq$ `b` ≤ 10.

Reals

`public static String E (double x)`

Same as `E (0, 6, x)`.

`public static String E (int fieldwidth, double x)`

Same as `E (fieldwidth, 6, x)`.

`public static String E (int fieldwidth, int precision, double x)`

Formats a double-precision number `x` like `%E` in C `printf`. The double argument is rounded and converted in the style `[-]d.dddE+-dd` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is 0, no decimal-point character appears. The exponent always contains at least two digits; if the value is zero, the exponent is `00`.

`public static String e (double x)`

Same as `e (0, 6, x)`.

`public static String e (int fieldwidth, double x)`

Same as `e (fieldwidth, 6, x)`.

`public static String e (int fieldwidth, int precision, double x)`

The same as `E`, except that `'e'` is used as the exponent character instead of `'E'`.

`public static String f (double x)`

Same as `f (0, 6, x)`.

`public static String f (int fieldwidth, double x)`

Same as `f (fieldwidth, 6, x)`.

`public static String f (int fieldwidth, int precision, double x)`

Formats the double-precision `x` into a string like `%f` in C `printf`. The argument is rounded and converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is explicitly 0, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

`public static String G (double x)`

Same as `G (0, 6, x)`.

`public static String G (int fieldwidth, double x)`

Same as `G (fieldwidth, 6, x)`.

`public static String G (int fieldwidth, int precision, double x)`

Formats the double-precision `x` into a string like `%G` in C `printf`. The argument is converted in style `%f` or `%E`. `precision` specifies the number of significant digits. If it is 0, it is treated as 1. Style `%E` is used if the exponent from its conversion is less than -4 or greater than or equal to `precision`. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

`public static String g (double x)`

Same as `g (0, 6, x)`.

`public static String g (int fieldwidth, double x)`

Same as `g (fieldwidth, 6, x)`.

`public static String g (int fieldwidth, int precision, double x)`

The same as `G`, except that `'e'` is used in the scientific notation.

`public static String format (int fieldwidth, int accuracy, int precision, double x)`

Returns a `String` containing `x`. Uses a total of at least `fieldwidth` positions (including the sign and point when they appear), `accuracy` digits after the decimal point and at least `precision` significant digits. `accuracy` and `precision` must be strictly smaller than

`fieldwidth`. The number is rounded if necessary. If there is not enough space to format the number in decimal notation with at least `precision` significant digits (`accuracy` or `fieldwidth` is too small), it will be converted to scientific notation with at least `precision` significant digits. In that case, `fieldwidth` is increased if necessary.

```
public static String format (Locale locale, int fieldwidth, int accuracy,
                             int precision, double x)
```

This method is equivalent to `format`, except it formats the given value for the locale `locale`.

```
public static String formatBase (int fieldwidth, int accuracy, int b,
                                 double x)
```

Converts x to a String representation in base b using formatting similar to the f methods. Uses a total of at least `fieldwidth` positions (including the sign and point when they appear) and `accuracy` digits after the decimal point. If `fieldwidth` is negative, the number is printed left-justified, otherwise right-justified. Restrictions: $2 \leq b \leq 10$ and $|x| < 2^{63}$.

Intervals

```
public static void formatWithError (int fieldwidth, int fieldwidtherr,
       int accuracy, int precision, double x, double error, String[] res)
```

Stores a string containing `x` into `res[0]`, and a string containing `error` into `res[1]`, both strings being formatted with the same notation. Uses a total of at least `fieldwidth` positions (including the sign and point when they appear) for `x`, `fieldwidtherr` positions for `error`, `accuracy` digits after the decimal point and at least `precision` significant digits. `accuracy` and `precision` must be strictly smaller than `fieldwidth`. The numbers are rounded if necessary. If there is not enough space to format `x` in decimal notation with at least `precision` significant digits (`accuracy` or `fieldwidth` are too small), it will be converted to scientific notation with at least `precision` significant digits. In that case, `fieldwidth` is increased if necessary, and the error is also formatted in scientific notation.

```
public static void formatWithError (int fieldwidth, int fieldwidtherr,
       int precision, double x, double error, String[] res)
```

Stores a string containing `x` into `res[0]`, and a string containing `error` into `res[1]`, both strings being formatted with the same notation. This calls `formatWithError` with the minimal accuracy for which the formatted string for `error` is non-zero. If `error` is 0, the accuracy is 0. If this minimal accuracy causes the strings to be formatted using scientific notation, this method increases the accuracy until the decimal notation can be used.

```
public static void formatWithError (Locale locale, int fieldwidth,
       int fieldwidtherr, int accuracy, int precision, double x,
       double error, String[] res)
```

This method is equivalent to `formatWithError`, except that it formats the given value and error for the locale `locale`.

```
public static void formatWithError (Locale locale, int fieldwidth,
       int fieldwidtherr, int precision, double x, double error,
       String[] res)
```

This method is equivalent to `formatWithError`, except that it formats the given value and error for the locale `locale`.

TableFormat

This class provides methods to format arrays and matrices into `String`s in different styles. This could be useful for printing arrays and subarrays, or for putting them in files for further treatment by other softwares such as *Mathematica*, *Matlab*, etc.

```
package umontreal.iro.lecuyer.util;

public class TableFormat
```

Formating styles

`public static final int PLAIN`

Plain text matrix printing style

`public static final int MATHEMATICA`

Mathematica matrix printing style

`public static final int MATLAB`

Matlab matrix printing style

Functions to convert arrays to `String`

```
public static String format (int V[], int n1, int n2, int k, int p)
```

Formats a `String` containing the elements `n1` to `n2` (inclusive) of table `V`, `k` elements per line, `p` positions per element. If $k = 1$, the array index will also appear on the left of each element, i.e., each line `i` will have the form `i V[i]`.

```
public static String format (double V[], int n1, int n2,
                             int k, int p1, int p2, int p3)
```

Similar to the previous method, but for an array of `double`'s. Gives at least `p1` positions per element, `p2` digits after the decimal point, and at least `p3` significant digits.

```
public static String format (double[][] Mat, int i1, int i2,
                             int j1, int j2, int w, int p,
                             int style, String Name)
```

Formats the submatrix with lines $\texttt{i1} \leq i \leq \texttt{i2}$ and columns $\texttt{j1} \leq j \leq \texttt{j2}$ of the matrix `Mat`, using the formatting style `style`. The elements are formated in `w` positions each, with a precision of `p` digits. `Name` provides an identifier for the submatrix. To be treated by `Matlab`, the returned string must be copied to a file with extension `.m`. If the file is named `poil.m`, for example, it can be accessed by calling `poil` in `Matlab`. For `Mathematica`, if the file is named `poil`, it will be read using `<< poil;`.

```
public static String format (int[][] Mat, int i1, int i2, int j1, int j2,
                             int w, int style, String Name)
```

Similar to the previous method, but for a matrix of `int`'s.

AbstractChrono

`AbstractChrono` is a class that acts as an interface to the system clock and calculates the CPU or system time consumed by parts of a program.

Every object of class `AbstractChrono` acts as an independent *stopwatch*. Several `AbstractChrono` objects can run at any given time. The method `init` resets the stopwatch to zero, `getSeconds`, `getMinutes` and `getHours` return its current reading, and `format` converts this reading to a `String`. The returned value includes the execution time of the method from `AbstractChrono`.

Below is an example of how it may be used. A stopwatch named `timer` is constructed (and initialized). When 2.1 seconds of CPU time have been consumed, the stopwatch is read and reset to zero. Then, after an additional 330 seconds (or 5.5 minutes) of CPU time, the stopwatch is read again and the value is printed to the output in minutes.

```
AbstractChrono timer = new Chrono();
    ⋮          (suppose 2.1 CPU seconds are used here.)
double t = timer.getSeconds();              // Here, t = 2.1
timer.init();
t = timer.getSeconds();                     // Here, t = 0.0
    ⋮          (suppose 330 CPU seconds are used here.)
t = timer.getMinutes();                     // Here, t = 5.5
System.out.println (timer.format());        // Prints: 0:5:30.00
```

```
package umontreal.iro.lecuyer.util;

public abstract class AbstractChrono
```

Timing functions

`public AbstractChrono()`

`public void init()`

Initializes this `AbstractChrono` to zero.

`public double getSeconds()`

Returns the CPU time in seconds used by the program since the last call to `init` for this `AbstractChrono`.

`public double getMinutes()`

Returns the CPU time in minutes used by the program since the last call to `init` for this `AbstractChrono`.

`public double getHours()`

Returns the CPU time in hours used by the program since the last call to `init` for this `AbstractChrono`.

`public String format()`

Converts the CPU time used by the program since its last call to `init` for this `AbstractChrono` to a `String` in the `HH:MM:SS.xx` format.

`public static String format (double time)`

Converts the time `time`, given in seconds, to a `String` in the `HH:MM:SS.xx` format.

SystemTimeChrono

Extends the `AbstractChrono` class to compute the total system time using Java's builtin `System.nanoTime`. The system can be used as a rough approximation of the CPU time taken by a program if no other tasks are executed on the host while the program is running.

```
package umontreal.iro.lecuyer.util;

public class SystemTimeChrono extends AbstractChrono
```

Constructor

`public SystemTimeChrono()`

Constructs a new chrono object and initializes it to zero.

GlobalCPUTimeChrono

Extends the `AbstractChrono` class to compute the global CPU time used by the Java Virtual Machine. This includes CPU time taken by any thread, including the garbage collector, class loader, etc.

Part of this class is implemented in the C language and the implementation is unfortunately operating system-dependent. The C functions for the current class have been compiled on a 32-bit machine running Linux. For a *platform-independent* CPU timer (valid only with Java–1.5 or later), one should use the class `ThreadCPUTimeChrono` which is programmed directly in Java.

```
package umontreal.iro.lecuyer.util;

public class GlobalCPUTimeChrono extends AbstractChrono
```

Constructor

`public GlobalCPUTimeChrono()`

Constructs a `Chrono` object and initializes it to zero.

ThreadCPUTimeChrono

Extends the `AbstractChrono` class to compute the CPU time for a single thread. It is available only under Java 1.5 which provides platform-independent facilities to get the CPU time for a single thread through management API.

Note that this chrono might not work properly on some systems running Linux because of a bug in Sun's implementation or Linux kernel. For instance, this class unexpectedly computes the global CPU time under Fedora Core 4, kernel 2.6.17 and JRE version 1.5.0-09. With Fedora Core 6, kernel 2.6.20, the function is working properly. As a result, one should not rely on this bug to get the global CPU time.

Note that the above bug does not prevent one from using this chrono to compute the CPU time for a single-threaded application. In that case, the global CPU time corresponds to the CPU time of the current thread.

Running timer fonctions when the associated thread is dead will return 0.

```
package umontreal.iro.lecuyer.util;

public class ThreadCPUTimeChrono extends AbstractChrono
```

Constructors

`public ThreadCPUTimeChrono()`

Constructs a `ThreadCPUTimeChrono` object associated with current thread and initializes it to zero.

`public ThreadCPUTimeChrono(Thread inThread)`

Constructs a `ThreadCPUTimeChrono` object associated with the given `Thread` variable and initializes it to zero.

Chrono

The `Chrono` class extends the `AbstractChrono` class and computes the CPU time for the current thread only. This is the simplest way to use chronos. Classes `AbstractChrono`, `SystemTimeChrono`, `GlobalCPUTimeChrono` and `ThreadCPUTimeChrono` provide different chronos implementations. See these classes to learn more about SSJ chronos, if problems appear with class `Chrono`.

```
package umontreal.iro.lecuyer.util;

public class Chrono extends AbstractChrono
```

Constructor

`public Chrono()`

Constructs a `Chrono` object and initializes it to zero.

Methods

`public static Chrono createForSingleThread ()`

Creates a `Chrono` instance adapted for a program using a single thread. Under Java 1.5, this method returns an instance of `ChronoSingleThread` which can measure CPU time for one thread. Under Java versions prior to 1.5, this returns an instance of this class. This method must not be used to create a timer for a multi-threaded program, because the obtained CPU times will differ depending on the used Java version.

ChronoSingleThread

This class is deprecated but kept for compatibility with older versions of SSJ. `Chrono` should be used instead of `ChronoSingleThread`. The `ChronoSingleThread` class extends the `AbstractChrono` class and computes the CPU time for the current thread only. This is the simplest way to use chronos. Classes `AbstractChrono`, `SystemTimeChrono`, `GlobalCPUTimeChrono` and `ThreadCPUTimeChrono` provide different chronos implementations (see these classes to learn more about SSJ chronos).

```
package umontreal.iro.lecuyer.util;


@Deprecated
public class ChronoSingleThread
```

Constructor

`public ChronoSingleThread()`

Constructs a `ChronoSingleThread` object and initializes it to zero.

TimeUnit

Represents a time unit for conversion of time durations. A time unit instance can be used to get information about the time unit and as a selector to perform conversions. Each time unit has a short name used when representing a time unit, a full descriptive name, and the number of hours corresponding to one unit.

```
package umontreal.iro.lecuyer.util;

public enum TimeUnit
```

enum values

`NANOSECOND`
: Represents a nanosecond which has short name `ns`.

`MICROSECOND`
: Represents a microsecond which has short name `us`.

`MILLISECOND`
: Represents a millisecond which has short name `ms`.

`SECOND`
: Represents a second which has short name `s`.

`MINUTE`
: Represents a minute which has short name `min`.

`HOUR`
: Represents an hour which has short name `h`.

`DAY`
: Represents a day which has short name `d`.

`WEEK`
: Represents a week which has short name `w`.

Methods

`public String getShortName()`

Returns the short name representing this unit in a string specifying a time duration.

`public String getLongName()`

Returns the long name of this time unit.

`public String toString()`

Calls `getLongName`.

`public double getHours()`

Returns this time unit represented in hours. This returns the number of hours corresponding to one unit.

```
public static double convert (double value, TimeUnit srcUnit,
                              TimeUnit dstUnit)
```

Converts `value` expressed in time unit `srcUnit` to a time duration expressed in `dstUnit` and returns the result of the conversion.

Systeme

This class provides a few tools related to the system or the computer.

```
package umontreal.iro.lecuyer.util;

public class Systeme
```

Methods

`public static String getHostName()`

Returns the name of the host computer.

`public static String getProcessInfo()`

Returns information about the running process: name, id, host name, date and time.

ArithmeticMod

This class provides facilities to compute multiplications of scalars, of vectors and of matrices modulo m. All algorithms are present in three different versions. These allow operations on `double`, `int` and `long`. The `int` and `long` versions work exactly like the `double` ones.

```
package umontreal.iro.lecuyer.util;

public class ArithmeticMod
```

Methods using `double`

```
public static double multModM (double a, double s, double c, double m)
```

Computes $(a \times s + c) \bmod m$. Where `m` must be smaller than 2^{35}. Works also if `s` or `c` are negative. The result is always positive (and thus always between 0 and `m` - 1).

```
public static void matVecModM (double A[][], double s[], double v[],
                               double m)
```

Computes the result of $\mathtt{A} \times \mathtt{s} \bmod m$ and puts the result in `v`. Where `s` and `v` are both column vectors. This method works even if `s` = `v`.

```
public static void matMatModM (double A[][], double B[][], double C[][],
                               double m)
```

Computes $\mathtt{A} \times \mathtt{B} \bmod m$ and puts the result in `C`. Works even if `A` = `C`, `B` = `C` or `A` = `B` = `C`.

```
public static void matTwoPowModM (double A[][], double B[][], double m,
                                  int e)
```

Computes $\mathtt{A}^{2^{\mathtt{e}}} \bmod m$ and puts the result in `B`. Works even if `A` = `B`.

```
public static void matPowModM (double A[][], double B[][], double m,
                               int c)
```

Computes $\mathtt{A}^{c} \bmod m$ and puts the result in `B`. Works even if `A` = `B`.

Methods using `int`

```
public static int multModM (int a, int s, int c, int m)
```

Computes $(a \times s + c) \bmod m$. Works also if `s` or `c` are negative. The result is always positive (and thus always between 0 and `m` - 1).

```
public static void matVecModM (int A[][], int s[], int v[], int m)
```

Exactly like `matVecModM` using `double`, but with `int` instead of `double`.

```
public static void matMatModM (int A[][], int B[][], int C[][], int m)
```

Exactly like `matMatModM` using `double`, but with `int` instead of `double`.

`public static void matTwoPowModM (int A[][], int B[][], int m, int e)`

Exactly like `matTwoPowModM` using `double`, but with `int` instead of `double`.

`public static void matPowModM (int A[][], int B[][], int m, int c)`

Exactly like `matPowModM` using `double`, but with `int` instead of `double`.

Methods using `long`

`public static long multModM (long a, long s, long c, long m)`

Computes $(a \times s + c) \bmod m$. Works also if `s` or `c` are negative. The result is always positive (and thus always between 0 and `m` - 1).

`public static void matVecModM (long A[][], long s[], long v[], long m)`

Exactly like `matVecModM` using `double`, but with `long` instead of `double`.

`public static void matMatModM (long A[][], long B[][], long C[][], long m)`

Exactly like `matMatModM` using `double`, but with `long` instead of `double`.

`public static void matTwoPowModM (long A[][], long B[][], long m, int e)`

Exactly like `matTwoPowModM` using `double`, but with `long` instead of `double`.

`public static void matPowModM (long A[][], long B[][], long m, int c)`

Exactly like `matPowModM` using `double`, but with `long` instead of `double`.

BitVector

This class implements vectors of bits and the operations needed to use them. The vectors can be of arbitrary length. The operations provided are all the binary operations available to the `int` and `long` primitive types in Java.

All bit operations are present in two forms: a normal form and a `self` form. The normal form returns a newly created object containing the result, while the `self` form puts the result in the calling object (`this`). The return value of the `self` form is the calling object itself. This is done to allow easier manipulation of the results, making it possible to chain operations.

```
package umontreal.iro.lecuyer.util;

public class BitVector implements Serializable, Cloneable
```

Constructors

`public BitVector (int length)`

Creates a new `BitVector` of length `length` with all its bits set to 0.

`public BitVector (int[] vect, int length)`

Creates a new `BitVector` of length `length` using the data in `vect`. Component `vect[0]` makes the 32 lowest order bits, with `vect[1]` being the 32 next lowest order bits, and so on. The normal bit order is then used to fill the 32 bits (the first bit is the lowest order bit and the last bit is largest order bit). Note that the sign bit is used as the largest order bit.

`public BitVector (int[] vect)`

Creates a new `BitVector` using the data in `vect`. The length of the `BitVector` is always equals to 32 times the length of `vect`.

`public BitVector (BitVector that)`

Creates a copy of the `BitVector that`.

Methods

`public Object clone()`

Creates a copy of the `BitVector`.

`public boolean equals (BitVector that)`

Verifies if two `BitVector`'s have the same length and the same data.

`public int size()`

Returns the length of the `BitVector`.

`public void enlarge (int size, boolean filling)`

Resizes the `BitVector` so that its length is equal to `size`. If the `BitVector` is enlarged, then the newly added bits are given the value 1 if `filling` is set to `true` and 0 otherwise.

`public void enlarge (int size)`

Resizes the `BitVector` so that its length is equal to `size`. Any new bit added is set to 0.

`public boolean getBool (int pos)`

Gives the value of the bit in position `pos`. If the value is 1, returns `true`; otherwise, returns `false`.

`public void setBool (int pos, boolean value)`

Sets the value of the bit in position `pos`. If `value` is equal to `true`, sets it to 1; otherwise, sets it to 0.

`public int getInt (int pos)`

Returns an `int` containing all the bits in the interval $[\texttt{pos} \times 32, \texttt{pos} \times 32 + 31]$.

`public String toString()`

Returns a string containing all the bits of the `BitVector`, starting with the highest order bit and finishing with the lowest order bit. The bits are grouped by groups of 8 bits for ease of reading.

`public BitVector not()`

Returns a `BitVector` which is the result of the `not` operator on the current `BitVector`. The `not` operator is equivalent to the `~` operator in Java and thus swap all bits (bits previously set to 0 become 1 and bits previously set to 1 become 0).

`public BitVector selfNot()`

Applies the `not` operator on the current `BitVector` and returns it.

`public BitVector xor (BitVector that)`

Returns a `BitVector` which is the result of the `xor` operator applied on `this` and `that`. The `xor` operator is equivalent to the `^` operator in Java. All bits which were set to 0 in one of the vector and to 1 in the other vector are set to 1. The others are set to 0. This is equivalent to the addition in modulo 2 arithmetic.

`public BitVector selfXor (BitVector that)`

Applies the `xor` operator on `this` with `that`. Stores the result in `this` and returns it.

`public BitVector and (BitVector that)`

Returns a `BitVector` which is the result of the `and` operator with both the `this` and `that` `BitVector`'s. The `and` operator is equivalent to the `&` operator in Java. Only bits which are set to 1 in both `this` and `that` are set to 1 in the result, all the others are set to 0.

`public BitVector selfAnd (BitVector that)`

Applies the `and` operator on `this` with `that`. Stores the result in `this` and returns it.

`public BitVector or (BitVector that)`

Returns a `BitVector` which is the result of the `or` operator with both the `this` and `that` `BitVector`'s. The `or` operator is equivalent to the `|` operator in Java. Only bits which are set to 0 in both `this` and `that` are set to to 0 in the result, all the others are set to 1.

`public BitVector selfOr (BitVector that)`

Applies the `or` operator on `this` with `that`. Stores the result in `this` and returns it.

`public BitVector shift (int j)`

Returns a `BitVector` equal to the original with all the bits shifted `j` positions to the right if `j` is positive, and shifted `j` positions to the left if `j` is negative. The new bits that appears to the left or to the right are set to 0. If `j` is positive, this operation is equivalent to the `>>>` operator in Java, otherwise, it is equivalent to the `<<` operator.

`public BitVector selfShift (int j)`

Shift all the bits of the current `BitVector` `j` positions to the right if `j` is positive, and `j` positions to the left if `j` is negative. The new bits that appears to the left or to the rigth are set to 0. Returns `this`.

`public boolean scalarProduct (BitVector that)`

Returns the scalar product of two `BitVector`'s modulo 2. It returns `true` if there is an odd number of bits with a value of 1 in the result of the `and` operator applied on `this` and `that`, and returns `false` otherwise.

BitMatrix

This class implements matrices of bits of arbitrary dimensions. Basic facilities for bits operations, multiplications and exponentiations are provided.

```
package umontreal.iro.lecuyer.util;

public class BitMatrix implements Serializable, Cloneable
```

Constructors

`public BitMatrix (int r, int c)`

Creates a new `BitMatrix` with `r` rows and `c` columns filled with 0's.

`public BitMatrix (BitVector[] rows)`

Creates a new `BitMatrix` using the data in `rows`. Each of the `BitVector` will be one of the rows of the `BitMatrix`.

`public BitMatrix (int[][] data, int r, int c)`

Creates a new `BitMatrix` with `r` rows and `c` columns using the data in `data`. Note that the orders of the bits for the rows are using the same order than for the `BitVector`. This does mean that the first bit is the lowest order bit of the last `int` in the row and the last bit is the highest order bit of the first `int` int the row.

`public BitMatrix (BitMatrix that)`

Copy constructor.

Methods

`public Object clone()`

Creates a copy of the `BitMatrix`.

`public boolean equals (BitMatrix that)`

Verifies that `this` and `that` are strictly identical. They must both have the same dimensions and data.

`public String toString()`

Creates a `String` containing all the data of the `BitMatrix`. The result is displayed in a matrix form, with each row being put on a different line. Note that the bit at (0,0) is at the upper left of the matrix, while the bit at (0) in a `BitVector` is the least significant bit.

`public String printData()`

Creates a `String` containing all the data of the `BitMatrix`. The data is displayed in the same format as are the `int[][]` in `Java` code. This allows the user to print the representation of

a `BitMatrix` to be put, directly in the source code, in the constructor `BitMatrix(int[][], int, int)`. The output is not designed to be human-readable.

`public int numRows()`

Returns the number of rows of the `BitMatrix`.

`public int numColumns()`

Returns the number of columns of the `BitMatrix`.

`public boolean getBool (int row, int column)`

Returns the value of the bit in the specified row and column. If the value is 1, return `true`. If it is 0, return `false`.

`public void setBool (int row, int column, boolean value)`

Changes the value of the bit in the specified row and column. If `value` is `true`, changes it to 1. If `value` is `false` changes it to 0.

`public BitMatrix transpose()`

Returns the transposed matrix. The rows and columns are interchanged.

`public BitMatrix not()`

Returns the `BitMatrix` resulting from the application of the `not` operator on the original `BitMatrix`. The effect is to swap all the bits of the `BitMatrix`, turning all 0 into 1 and all 1 into 0.

`public BitMatrix and (BitMatrix that)`

Returns the `BitMatrix` resulting from the application of the `and` operator on the original `BitMatrix` and `that`. Only bits which were at 1 in both `BitMatrix` are set at 1 in the result. All others are set to 0.

`public BitMatrix or (BitMatrix that)`

Returns the `BitMatrix` resulting from the application of the `or` operator on the original `BitMatrix` and `that`. Only bits which were at 0 in both `BitMatrix` are set at 0 in the result. All others are set to 1.

`public BitMatrix xor (BitMatrix that)`

Returns the `BitMatrix` resulting from the application of the `xor` operator on the original `BitMatrix` and `that`. Only bits which were at 1 in only one of the two `BitMatrix` are set at 1 in the result. All others are set to 0.

`public BitVector multiply (BitVector vect)`

Multiplies the column `BitVector` by a `BitMatrix` and returns the result. The result is $A \times v$, where A is the `BitMatrix`, and v is the `BitVector`.

`public int multiply (int vect)`

Multiplies `vect`, seen as a column `BitVector`, by a `BitMatrix`. (See `BitVector` to see the conversion between `int` and `BitVector`.) The result is $A \times v$, where A is the `BitMatrix`, and v is the `BitVector`.

`public BitMatrix multiply (BitMatrix that)`

Multiplies two `BitMatrix`'s together. The result is $A \times B$, where A is the `this BitMatrix` and B is the `that BitMatrix`.

`public BitMatrix power (long p)`

Raises the `BitMatrix` to the power `p`.

`public BitMatrix power2e (int e)`

Raises the `BitMatrix` to power $2^{\texttt{e}}$.

Nested Class

`public class IncompatibleDimensionException extends RuntimeException`

Runtime exception raised when the dimensions of the `BitMatrix` are not appropriate for the operation.

DMatrix

This class implements a few methods for matrix calculations with `double` numbers.

```
package umontreal.iro.lecuyer.util;
   import cern.colt.matrix.*;


public class DMatrix
```

Constructors

`public DMatrix (int r, int c)`

Creates a new `DMatrix` with `r` rows and `c` columns.

`public DMatrix (double[][] data, int r, int c)`

Creates a new `DMatrix` with `r` rows and `c` columns using the data in `data`.

`public DMatrix (DMatrix that)`

Copy constructor.

Methods

`public static void CholeskyDecompose (double[][] M, double[][] L)`

Given a symmetric positive-definite matrix M, performs the Cholesky decomposition of M and returns the result as a lower triangular matrix L, such that $M = LL^T$.

`public static DoubleMatrix2D CholeskyDecompose (DoubleMatrix2D M)`

Given a symmetric positive-definite matrix M, performs the Cholesky decomposition of M and returns the result as a lower triangular matrix L, such that $M = LL^T$.

```
public static void PCADecompose (double[][] M, double[][] A,
                                 double[] lambda)
```

Computes the principal components decomposition $M = U\Lambda U^{\mathsf{t}}$ by using the singular value decomposition of matrix M. Puts the eigenvalues, which are the diagonal elements of matrix Λ, sorted by decreasing size, in vector `lambda`, and puts matrix $A = U\sqrt{\Lambda}$ in `A`.

```
public static DoubleMatrix2D PCADecompose (DoubleMatrix2D M,
                                           double[] lambda)
```

Computes the principal components decomposition $M = U\Lambda U^{\mathsf{t}}$ by using the singular value decomposition of matrix M. Puts the eigenvalues, which are the diagonal elements of matrix Λ, sorted by decreasing size, in vector `lambda`. Returns matrix $A = U\sqrt{\Lambda}$.

`public static double[] solveLU (double[][] A, double[] b)`

Solves the matrix equation $Ax = b$ using LU decomposition. A is a square matrix, b and x are vectors. Returns the solution x.

`public static void solveTriangular (DoubleMatrix2D U, DoubleMatrix2D B, DoubleMatrix2D X)`

Solve the triangular matrix equation $UX = B$ for X. U is a square upper triangular matrix. B and X must have the same number of columns.

`public static double[][] exp (double[][] A)`

Similar to `exp(A)`.

`public static DoubleMatrix2D exp (final DoubleMatrix2D A)`

Returns e^A, the exponential of the square matrix A. The scaling and squaring method [3] is used with Padé approximants to compute the exponential.

`public static DoubleMatrix2D expBidiagonal (final DoubleMatrix2D A)`

Returns e^A, the exponential of the *bidiagonal* square matrix A. The only non-zero elements of A are on the diagonal and on the first superdiagonal. This method is faster than `exp(A)` because of the special form of A.

`public static DoubleMatrix1D expBidiagonal (final DoubleMatrix2D A, final DoubleMatrix1D b)`

Computes $c = e^A b$, where e^A is the exponential of the *bidiagonal* square matrix A. The only non-zero elements of A are on the diagonal and on the first superdiagonal. Uses the scaling and squaring method [3] with Padé approximants. Returns c.

`public static DoubleMatrix2D expmiBidiagonal (final DoubleMatrix2D A)`

Computes $e^A - I$, where e^A is the exponential of the *bidiagonal* square matrix A. The only non-zero elements of A are on the diagonal and on the first superdiagonal. Uses the scaling and squaring method [5, 3] with Padé approximants. Returns $e^A - I$.

`public static DoubleMatrix1D expmiBidiagonal (final DoubleMatrix2D A, final DoubleMatrix1D b)`

Computes $c = (e^A - I)b$, where e^A is the exponential of the *bidiagonal* square matrix A. The only non-zero elements of A are on the diagonal and on the first superdiagonal. Uses the scaling and squaring method [5, 3] with a Taylor expansion. Returns c.

`public static void copy (double[][] M, double[][] R)`

Copies the matrix M into R.

`public static String toString(double[][] M)`

Returns matrix M as a string. It is displayed in matrix form, with each row on a line.

`public String toString()`

Creates a `String` containing all the data of the `DMatrix`. The result is displayed in matrix form, with each row on a line.

`public int numRows()`

Returns the number of rows of the `DMatrix`.

`public int numColumns()`

Returns the number of columns of the `DMatrix`.

`public double get (int row, int column)`

Returns the matrix element in the specified row and column.

`public void set (int row, int column, double value)`

Sets the value of the element in the specified row and column.

`public DMatrix transpose()`

Returns the transposed matrix. The rows and columns are interchanged.

RootFinder

This class provides methods to solve non-linear equations.

```
package umontreal.iro.lecuyer.util;
   import umontreal.iro.lecuyer.functions.MathFunction;


public class RootFinder
```

Methods

```
public static double brentDekker (double a, double b,
                                  MathFunction f, double tol)
```

Computes a root x of the function in `f` using the Brent-Dekker method. The interval $[a, b]$ must contain the root x. The calculations are done with an approximate relative precision `tol`. Returns x such that $f(x) = 0$.

```
public static double bisection (double a, double b,
                                MathFunction f, double tol)
```

Computes a root x of the function in `f` using the *bisection* method. The interval $[a, b]$ must contain the root x. The calculations are done with an approximate relative precision `tol`. Returns x such that $f(x) = 0$.

MultivariateFunction

Represents a function of multiple variables. This interface specifies a method `evaluate` that computes a $g(\mathbf{x})$ function, where $\mathbf{x} = (x_0, \ldots, x_{d-1}) \in \mathbb{R}^d$. It also specifies a method `evaluateGradient` for computing its gradient $\nabla g(\mathbf{x})$.

The dimension d can be fixed or variable. When d is fixed, the methods specified by this interface always take the same number of arguments. This is the case, for example, with a ratio of two variables. When d is variable, the implementation can compute the function for a vector $\mathbf{x}$ of any length. This can happen for a product or sum of variables.

The methods of this interface take a variable number of arguments to accomodate the common case of fixed dimension with more convenience; the programmer can call the method without creating an array. For the generic case, however, one can replace the arguments with an array.

```
package umontreal.iro.lecuyer.util;


public interface MultivariateFunction
```

```
   public int getDimension();
```

Returns d, the dimension of the function computed by this implementation. If the dimension is not fixed, this method must return a negative value.

```
   public double evaluate (double... x);
```

Computes the function $g(\mathbf{x})$ for the vector `x`. The length of the given array must correspond to the dimension of this function. The method must compute and return the result of the function without modifying the elements in `x` since the array can be reused for further computation.

```
   public double evaluateGradient (int i, double... x);
```

Computes $\partial g(\mathbf{x})/\partial x_i$, the derivative of $g(\mathbf{x})$ with respect to x_i. The length of the given array must correspond to the dimension of this function. The method must compute and return the result of the derivative without modifying the elements in `x` since the array can be reused for further computations, e.g., the gradient $\nabla g(\mathbf{x})$.

RatioFunction

Represents a function computing a ratio of two values.

```
package umontreal.iro.lecuyer.util;

public class RatioFunction implements MultivariateFunction
```

Constructors

`public RatioFunction()`

Constructs a new ratio function.

`public RatioFunction (double zeroOverZero)`

Constructs a new ratio function that returns `zeroOverZero` for the special case of 0/0. See the `getZeroOverZeroValue` method for more information. The default value of `zeroOverZero` is `Double.NaN`.

Methods

`public double getZeroOverZeroValue()`

Returns the value returned by `evaluate` in the case where the 0/0 function is calculated. The default value for 0/0 is `Double.NaN`.

Generally, 0/0 is undefined, and therefore associated with the `Double.NaN` constant, meaning *not-a-number*. However, in certain applications, it can be defined differently to accomodate some special cases. For exemple, in a queueing system, if there are no arrivals, no customers are served, lost, queued, etc. As a result, many performance measures of interest turn out to be 0/0. Specifically, the loss probability, i.e., the ratio of lost customers over the number of arrivals, should be 0 if there is no arrival; in this case, 0/0 means 0. On the other hand, the service level, i.e., the fraction of customers waiting less than a fixed threshold, could be fixed to 1 if there is no arrival.

`public void setZeroOverZeroValue (double zeroOverZero)`

Sets the value returned by `evaluate` for the undefined function 0/0 to `zeroOverZero`. See `getZeroOverZeroValue` for more information.

Misc

This class provides miscellaneous functions that are hard to classify. Some may be moved to another class in the future.

```
package umontreal.iro.lecuyer.util;

public class Misc
```

Methods

`public static double quickSelect (double[] A, int n, int k)`

Returns the k^{th} smallest item of the array A of size n. Array A is unchanged by the method. Restriction: $1 \leq k \leq n$.

`public static int quickSelect (int[] A, int n, int k)`

Returns the k^{th} smallest item of the array A of size n. Array A is unchanged by the method. Restriction: $1 \leq k \leq n$.

`public static double getMedian (double[] A, int n)`

Returns the median of the first n elements of array A.

`public static double getMedian (int[] A, int n)`

Returns the median of the first n elements of array A.

`public static int getTimeInterval (double[] times, int start, int end, double t)`

Returns the index of the time interval corresponding to time `t`. Let $t_0 \leq \cdots \leq t_n$ be simulation times stored in a subset of `times`. This method uses binary search to determine the smallest value i for which $t_i \leq t < t_{i+1}$, and returns i. The value of t_i is stored in `times[start+i]` whereas n is defined as `end - start`. If $t < t_0$, this returns -1. If $t \geq t_n$, this returns n. Otherwise, the returned value is greater than or equal to 0, and smaller than or equal to $n-1$. `start` and `end` are only used to set lower and upper limits of the search in the `times` array; the index space of the returned value always starts at 0. Note that if the elements of `times` with indices `start`, ..., `end` are not sorted in non-decreasing order, the behavior of this method is undefined.

`public static void interpol (int n, double[] X, double[] Y, double[] C)`

Computes the Newton interpolating polynomial. Given the $n+1$ real distinct points (x_0, y_0), $(x_1, y_1), \ldots, (x_n, y_n)$, with `X[i]` $= x_i$, `Y[i]` $= y_i$, this function computes the $n+1$ coefficients `C[i]` $= c_i$ of the Newton interpolating polynomial $P(x)$ of degree n passing through these points, i.e. such that $y_i = P(x_i)$, given by

$$P(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}). \quad (3)$$

`public static double evalPoly (int n, double[] X, double[] C, double z)`

Given n, X and C as described in `interpol(n, X, Y, C)`, this function returns the value of the interpolating polynomial $P(z)$ evaluated at z (see eq. 3).

`public static double evalPoly (double[] C, int n, double x)`

Evaluates the polynomial $P(x)$ of degree n with coefficients $c_j =$ `C[j]` at x:

$$P(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \tag{4}$$

JDBCManager

This class provides some facilities to connect to a SQL database and to retrieve data stored in it. JDBC provides a standardized interface for accessing a database independently of a specific database management system (DBMS). The user of JDBC must create a `Connection` object used to send SQL queries to the underlying DBMS, but the creation of the connection adds a DBMS-specific portion in the application. This class helps the developer in moving the DBMS-specific information out of the source code by storing it in a properties file. The methods in this class can read such a properties file and establish the JDBC connection. The connection can be made by using a `DataSource` obtained through a JNDI server, or by a JDBC URI associated with a driver class. Therefore, the properties used to connect to the database must be a JNDI name (`jdbc.jndi-name`), or a driver to load (`jdbc.driver`) with the URI of a database (`jdbc.uri`).

```
jdbc.driver=com.mysql.jdbc.Driver
```

```
jdbc.uri=jdbc:mysql://mysql.iro.umontreal.ca/database?user=foo&password=bar
```

The connection is established using the `connectToDatabase` method. Shortcut methods are also available to read the properties from a file or a resource before establishing the connection. This class also provides shortcut methods to read data from a database and to copy the data into Java arrays.

```
package umontreal.iro.lecuyer.util;


public class JDBCManager
```

Methods

```
public static Connection connectToDatabase (Properties prop)
         throws SQLException
```

Connects to the database using the properties `prop` and returns the an object representing the connection. The properties stored in `prop` must be a JNDI name (`jdbc.jndi-name`), or the name of a driver (`jdbc.driver`) to load and the URI of the database (`jdbc.uri`). When a JNDI name is given, this method constructs a context using the nullary constructor of `InitialContext`, uses the context to get a `DataSource` object, and uses the data source to obtain a connection. This method assumes that JNDI is configured correctly; see the class `InitialContext` for more information about configuring JNDI. If no JNDI name is specified, the method looks for a JDBC URI. If a driver class name is specified along with the URI, the corresponding driver is loaded and registered with the JDBC `DriverManager`. The driver manager is then used to obtain the connection using the URI. This method throws an `SQLException` if the connection failed and an `IllegalArgumentException` if the properties do not contain the required values.

```
public static Connection connectToDatabase (InputStream is)
         throws IOException, SQLException
```

Returns a connection to the database using the properties read from stream `is`. This method loads the properties from the given stream, and calls `connectToDatabase` to establish the connection.

```
public static Connection connectToDatabase (URL url)
        throws IOException, SQLException
```

Equivalent to `connectToDatabase (url.openStream())`.

```
public static Connection connectToDatabase (File file)
        throws IOException, SQLException
```

Equivalent to `connectToDatabase (new FileInputStream (file))`.

```
public static Connection connectToDatabase (String fileName)
        throws IOException, SQLException
```

Equivalent to `connectToDatabase (new FileInputStream (fileName))`.

```
public static Connection connectToDatabaseFromResource (String resource)
        throws IOException, SQLException
```

Uses `connectToDatabase` with the stream obtained from the resource `resource`. This method searches the file `resource` on the class path, opens the first resource found, and extracts properties from it. It then uses `connectToDatabase` to establish the connection.

```
public static double[] readDoubleData (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of double-precision values. This method uses the statement `stmt` to execute the given query, and assumes that the first column of the result set contains double-precision values. Each row of the result set then becomes an element of an array of double-precision values which is returned by this method. This method throws an `SQLException` if the query is not valid.

```
public static double[] readDoubleData (Connection connection,
                                       String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of double-precision values. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readDoubleData`, which returns an array of double-precision values.

```
public static double[] readDoubleData (Statement stmt, String table,
                                       String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readDoubleData (stmt, "SELECT column FROM table")`.

```
public static double[] readDoubleData (Connection connection,
                                       String table, String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readDoubleData (connection, "SELECT column FROM table")`.

```
public static int[] readIntData (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of integers. This method uses the statement `stmt` to execute the given query, and assumes that the first column of the result

set contains integer values. Each row of the result set then becomes an element of an array of integers which is returned by this method. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static int[] readIntData (Connection connection, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of integers. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readIntData`, which returns an array of integers.

```
public static int[] readIntData (Statement stmt, String table,
                                 String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readIntData (stmt, "SELECT column FROM table")`.

```
public static int[] readIntData (Connection connection, String table,
                                 String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readIntData (connection, "SELECT column FROM table")`.

```
public static Object[] readObjectData (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of objects. This method uses the statement `stmt` to execute the given query, and extracts values from the first column of the obtained result set by using the `getObject` method. Each row of the result set then becomes an element of an array of objects which is returned by this method. The type of the objects in the array depends on the column type of the result set, which depends on the database and query. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static Object[] readObjectData (Connection connection,
                                       String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of objects. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readObjectData`, which returns an array of integers.

```
public static Object[] readObjectData (Statement stmt,
                                       String table, String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readObjectData (stmt, "SELECT column FROM table")`.

```
public static Object[] readObjectData (Connection connection,
                                       String table, String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readObjectData (connection, "SELECT column FROM table")`.

```
public static double[][] readDoubleData2D (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of double-precision values. This method uses the statement `stmt` to execute the given query, and assumes that the columns of the result set contain double-precision values. Each row of the result set then becomes a row of a 2D array of double-precision values which is returned by this method. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static double[][] readDoubleData2D (Connection connection,
                                           String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of double-precision values. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readDoubleData2D`, which returns a 2D array of double-precision values.

```
public static double[][] readDoubleData2DTable (Statement stmt,
                                                String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readDoubleData2D (stmt, "SELECT * FROM table")`.

```
public static double[][] readDoubleData2DTable (Connection connection,
                                                String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readDoubleData2D (connection, "SELECT * FROM table")`.

```
public static int[][] readIntData2D (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of integers. This method uses the statement `stmt` to execute the given query, and assumes that the columns of the result set contain integers. Each row of the result set then becomes a row of a 2D array of integers which is returned by this method. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static int[][] readIntData2D (Connection connection, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of integers. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readIntData2D`, which returns a 2D array of integers.

```
public static int[][] readIntData2DTable (Statement stmt, String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readIntData2D (stmt, "SELECT * FROM table")`.

```
public static int[][] readIntData2DTable (Connection connection,
                                          String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readIntData2D (connection, "SELECT * FROM table")`.

```
public static Object[][] readObjectData2D (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of objects. This method uses the statement `stmt` to execute the given query, and extracts values from the obtained result set by using the `getObject` method. Each row of the result set then becomes a row of a 2D array of objects which is returned by this method. The type of the objects in the 2D array depends on the column types of the result set, which depend on the database and query. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static Object[][] readObjectData2D (Connection connection,
                                           String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of integers. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readObjectData2D`, which returns a 2D array of integers.

```
public static Object[][] readObjectData2DTable (Statement stmt,
                                                String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readObjectData2D (stmt, "SELECT * FROM table")`.

```
public static Object[][] readObjectData2DTable (Connection connection,
                                                String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readObjectData2D (connection, "SELECT * FROM table")`.

ClassFinder

Utility class used to convert a simple class name to a fully qualified class object. The `Class` class can be used to obtain information about a class (its name, its fields, methods, constructors, etc.), and to construct objects, even if the exact class is known at runtime only. It provides a `forName` static method converting a string to a `Class`, but the given string must be a fully qualified name.

Sometimes, configuration files may need to contain Java class names. After they are extracted from the file, these class names are given to `forName` to be converted into `Class` objects. Unfortunately, only fully qualified class names will be accepted as input, which clutters configuration files, especially if long package names are used. This class permits the definition of a set of import declarations in a way similar to the Java Language Specification [2]. It provides methods to convert a simple class name to a `Class` object and to generate a simple name from a `Class` object, based on the import rules.

The first step for using a class finder is to construct an instance of this class. Then, one needs to retrieve the initially empty list of import declarations by using `getImports`, and update it with the actual import declarations. Then, the method `findClass` can find a class using the import declarations. For example, the following code retrieves the class object for the `List` class in package `java.util`

```
ClassFinder cf = new ClassFinder();
cf.getImports().add ("java.util.*");
Class<?> listClass = cf.findClass ("List");
```

```
package umontreal.iro.lecuyer.util;


public class ClassFinder implements Cloneable, java.io.Serializable
```

`private List<List<String>> imports`

Contains the saved import lists. Each element of this list is a nested `List` containing `String`'s, each string containing the fully qualified name of an imported package or class.

`public ClassFinder()`

Constructs a new class finder with an empty list of import declarations.

`public List<String> getImports()`

Returns the current list of import declarations. This list may contain only `String`'s of the form `java.class.name` or `java.package.name.*`.

`public void saveImports()`

Saves the current import list on the import stack. This method makes a copy of the list returned by `getImports` and puts it on top of a stack to be restored later by `restoreImports`.

`public void restoreImports()`

Restores the list of import declarations. This method removes the last list of import declarations from the stack. If the stack contains only one list, this list is cleared.

```
public Class<?> findClass (String name) throws
   ClassNotFoundException, NameConflictException
```

Tries to find the class corresponding to the simple name `name`. The method first considers the argument as a fully qualified class name and calls `forName (name)`. If the class cannot be found, it considers the argument as a simple name. A simple name refers to a class without specifying the package declaring it. To convert simple names to qualified names, the method iterates through all the strings in the list returned by `getImports`, applying the same rules as a Java compiler to resolve the class name. However, if an imported package or class does not exist, it will be ignored whereas the compiler would stop with an error.

For the class with simple name `name` to be loaded, it must be imported explicitly (single-type import) or one of the imported packages must contain it (type import on-demand). If the class with name `name` is imported explicitly, this import declaration has precedence over any imported packages. If several import declaration match the given simple name, e.g., if several fully qualified names with the same simple name are imported, or if a class with simple name `name` exists in several packages, a `NameConflictException` is thrown.

```
public String getSimpleName (Class<?> cls)
```

Returns the simple name of the class `cls` that can be used when the imports contained in this class finder are used. For example, if `java.lang.String.class` is given to this method, `String` is returned if `java.lang.*` is among the import declarations.

Note: this method does not try to find name conflicts. This operation is performed by `findClass` only. For example, if the list of imported declarations contains `foo.bar.*` and `test.Foo`, and the simple name for `test.Foo` is queried, the method returns `Foo` even if the package `foo.bar` contains a `Foo` class.

```
public ClassFinder clone()
```

Clones this class finder, and copies its lists of import declarations.

NameConflictException

This exception is thrown by a `ClassFinder` when two or more fully qualified class names can be associated with a simple class name.

```
package umontreal.iro.lecuyer.util;

public class NameConflictException extends Exception
```

Constructors

`public NameConflictException()`

Constructs a new name conflict exception.

`public NameConflictException (String message)`

Constructs a new name conflict exception with message `message`.

```
public NameConflictException (ClassFinder finder, String name,
                              String message)
```

Constructs a new name conflict exception with class finder `finder`, simple name `name`, and message `message`.

Methods

`public ClassFinder getClassFinder()`

Returns the class finder associated with this exception.

`public String getName()`

Returns the simple name associated with this exception.

Introspection

Provides utility methods for introspection using Java Reflection API.

```
package umontreal.iro.lecuyer.util;


public class Introspection
```

`public static Method[] getMethods (Class<?> c)`

Returns all the methods declared and inherited by a class. This is similar to `getMethods` except that it enumerates non-public methods as well as public ones. This method uses `getDeclaredMethods` to get the declared methods of `c`. It also gets the declared methods of superclasses. If a method is defined in a superclass and overriden in a subclass, only the overriden method will be in the returned array.

Note that since this method uses `getDeclaredMethods`, it can throw a `SecurityException` if a security manager is present.

`public static boolean sameSignature (Method m1, Method m2)`

Determines if two methods `m1` and `m2` share the same signature. For the signature to be identical, methods must have the same number of parameters and the same parameter types.

`public static Field[] getFields (Class<?> c)`

Returns all the fields declared and inherited by a class. This is similar to `getFields` except that it enumerates non-public fields as well as public ones. This method uses `getDeclaredFields` to get the declared fields of `c`. It also gets the declared fields of superclasses and implemented interfaces.

Note that since this method uses `getDeclaredFields`, it can throw a `SecurityException` if a security manager is present.

```
public static Method getMethod (Class<?> c, String name, Class[] pt)
                                throws NoSuchMethodException
```

This is like `getMethod`, except that it can return non-public methods.

```
public static Field getField (Class<?> c, String name)
                              throws NoSuchFieldException
```

This is like `getField`, except that it can return non-public fields.

Note that since this method uses `getDeclaredField`, it can throw a `SecurityException` if a security manager is present.

`public static String getFieldName (Object val)`

Returns the field name corresponding to the value of an enumerated type `val`. This method gets the class of `val` and scans its fields to find a public static and final field containing `val`. If such a field is found, its name is returned. Otherwise, `null` is returned.

`public static <T> T valueOf (Class<T> cls, String name)`

Returns the field of class `cls` corresponding to the name `name`. This method looks for a public, static, and final field with name `name` and returns its value. If no appropriate field can be found, an `IllegalArgumentException` is thrown.

`public static <T> T valueOfIgnoreCase (Class<T> cls, String name)`

Similar to `valueOf (cls, name)`, with case insensitive field name look-up. If `cls` defines several fields with the same case insensitive name `name`, an `IllegalArgumentException` is thrown.

TransformingList

Represents a list that dynamically transforms the elements of another list. This abstract class defines a list containing an inner list of elements of a certain type, and provides facilities to convert these inner elements to outer elements of another type. A concrete subclass simply needs to provide methods for converting between the inner and the outer types.

```
package umontreal.iro.lecuyer.util;

public abstract class TransformingList<OE,IE> extends AbstractList<OE>
```

`public TransformingList (List<IE> fromList)`

Creates a new transforming list wrapping the inner list `fromList`.

`public abstract OE convertFromInnerType (IE e)`

Converts an element in the inner list to an element of the outer type.

`public abstract IE convertToInnerType (OE e)`

Converts an element of the outer type to an element for the inner list.

DoubleArrayComparator

An implementation of `java.util.Comparator` which compares two `double` arrays by comparing their `i`-*th* element, where `i` is given in the constructor. Method `compare(d1, d2)` returns −1, 0, or 1 depending on whether `d1[i]` is less than, equal to, or greater than `d2[i]`.

```
package umontreal.iro.lecuyer.util;

public class DoubleArrayComparator implements Comparator<double[]>
```

Constructor

`public DoubleArrayComparator (int i)`

Constructs a comparator, where `i` is the index used for the comparisons.

Methods

`public int compare (double[] d1, double[] d2)`

Returns −1, 0, or 1 depending on whether `d1[i]` is less than, equal to, or greater than `d2[i]`.

Overview of package `util.io`

This package provides tools for exporting data to text and binary files, as well as for importing data from files.

Each of the `write()` methods takes a *field label* as their first argument. This label can always be set to `null`, in which case an anonymous field will be written. The `write()` methods that take one-dimensional array argument can also take an additional integer argument, for convenience, to specify the number of elements to write in the array.

For a quick start, consult the following examples and the documentation for `DataWriter` and `DataReader`, as well as the constructors of implementing classes (`TextDataWriter`, `BinaryDataWriter` and `BinaryDataReader`).

Example of how to write data to a file:

```
public static void writerExample() throws IOException {
   String filename = "test.dat";
   DataWriter out = new BinaryDataWriter(filename);
   out.write("zero", 0);
   out.write("zerotxt", "ZERO");
   out.write("n", new int[]{1,2,3,4,5});
   out.write("pi", Math.PI);
   out.write("str", new String[]{"text1", "text2"});
   out.write("real", new double[]{2.5, 3.7, 8.9});
   out.write("real2", new float[]{2.5f, 3.7f, 8.9f});
   out.write(null, 24);
   out.write(null, 39);
   out.write(null, 116);
   out.close();
}
```

Example of how to read data from a file — specific fields:

```
public static void readerExample1() throws IOException {
   String filename = "test.dat";
   DataReader in = new BinaryDataReader(filename);

   // read double field labeled "pi"
   System.out.println("[pi] (double) " + in.readField("pi").asDouble());

   // read integer-array field labeled "n"
   int[] n = in.readIntArray("n");
   System.out.print("[n] (int[]) ");
   for (int i = 0; i < n.length; i++)
      System.out.print(" " + n[i]);
   System.out.println();

   in.close();
}
```

Example of how to read data from a file — list all fields:

```
public static void readerExample2() throws IOException {
   String filename = "test.dat";
   DataReader in = new BinaryDataReader(filename);

   Map<String,DataField> fields = in.readAllFields();
   in.close();

   // sort keys
   Set<String> allKeys = new TreeSet<String>(fields.keySet());

   for (String key : allKeys) {
      System.out.print("[" + key + "]");
      DataField d = fields.get(key);

      if (d.isString())
         System.out.print(" (String) " + d.asString());

      if (d.isInt())
         System.out.print(" (int) " + d.asInt());

      if (d.isFloat())
         System.out.print(" (float) " + d.asFloat());

      if (d.isDouble())
         System.out.print(" (double) " + d.asDouble());

      if (d.asStringArray() != null) {
         System.out.print(" (String[]) ");
         String[] a = d.asStringArray();
         for (int i = 0; i < a.length; i++)
            System.out.print(" " + a[i]);
      }

      if (d.asIntArray() != null) {
         System.out.print(" (int[]) ");
         int[] a = d.asIntArray();
         for (int i = 0; i < a.length; i++)
            System.out.print(" " + a[i]);
      }

      if (d.asFloatArray() != null) {
         System.out.print(" (float[]) ");
         float[] a = d.asFloatArray();
         for (int i = 0; i < a.length; i++)
            System.out.print(" " + a[i]);
      }

      if (d.asDoubleArray() != null) {
         System.out.print(" (double[]) ");
         double[] a = d.asDoubleArray();
         for (int i = 0; i < a.length; i++)
            System.out.print(" " + a[i]);
      }

      System.out.println();
   }
}
```

DataWriter

Data writer interface.

```
package umontreal.iro.lecuyer.util.io;

public interface DataWriter
```

Writing atomic data

`public void write (String label, String s) throws IOException;`
Writes an atomic string field. Writes an anonymous field if `label` is `null`.

`public void write (String label, int a) throws IOException;`
Writes an atomic 32-bit integer (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float a) throws IOException;`
Writes an atomic 32-bit float (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double a) throws IOException;`
Writes an atomic 64-bit double (big endian). Writes an anonymous field if `label` is `null`.

Writing one-dimensional arrays

`public void write (String label, String[] a) throws IOException;`
Writes a one-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, String[] a, int n) throws IOException;`
Writes the first `n` elements of a one-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, int[] a) throws IOException;`
Writes a one-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, int[] a, int n) throws IOException;`
Writes the first `n` elements of a one-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[] a) throws IOException;`
Writes a one-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[] a, int n) throws IOException;`
Writes the first `n` elements of a one-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[] a) throws IOException;`

Writes a one-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[] a, int n) throws IOException;`

Writes the first `n` elements of a one-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

Writing two-dimensional arrays

`public void write (String label, String[][] a) throws IOException;`

Writes a two-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, int[][] a) throws IOException;`

Writes a two-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[][] a) throws IOException;`

Writes a two-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[][] a) throws IOException;`

Writes a two-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

Other methods

`public void close() throws IOException;`

Flushes any pending data and closes the output stream.

AbstractDataWriter

This abstract class implements shared functionality for data writers.

```
package umontreal.iro.lecuyer.util.io;

public abstract class AbstractDataWriter implements DataWriter
```

Writing one-dimensional arrays

`public void write (String label, String[] a) throws IOException`

Writes a one-dimensional array of strings. If `label` is `null`, writes an anonymous field.

`public void write (String label, int[] a) throws IOException`

Writes a one-dimensional array of 32-bit integers (big endian). If `label` is `null`, writes an anonymous field.

`public void write (String label, float[] a) throws IOException`

Writes a one-dimensional array of 32-bit floats (big endian). If `label` is `null`, writes an anonymous field.

`public void write (String label, double[] a) throws IOException`

Writes a one-dimensional array of 64-bit doubles (big endian). If `label` is `null`, writes an anonymous field.

CachedDataWriter

This abstract class implements shared functionality for data writers that store all fields in memory before outputing them with `close`.

```
package umontreal.iro.lecuyer.util.io;

public abstract class CachedDataWriter extends AbstractDataWriter
```

Constructor

`public CachedDataWriter()`
Class constructor.

Writing atomic data

`public void write (String label, String s) throws IOException`
Writes an atomic string field. Writes an anonymous field if `label` is `null`.

`public void write (String label, int a) throws IOException`
Writes an atomic 32-bit integer (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float a) throws IOException`
Writes an atomic 32-bit float (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double a) throws IOException`
Writes an atomic 64-bit double (big endian). Writes an anonymous field if `label` is `null`.

Writing one-dimensional arrays

`public void write (String label, String[] a, int n) throws IOException`
Writes the first `n` elements of a one-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, int[] a, int n) throws IOException`
Writes the first `n` elements of a one-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[] a, int n) throws IOException`
Writes the first `n` elements of a one-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[] a, int n) throws IOException`
Writes the first `n` elements of a one-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

Writing two-dimensional arrays

`public void write (String label, String[][] a) throws IOException`

Writes a two-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, int[][] a) throws IOException`

Writes a two-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[][] a) throws IOException`

Writes a two-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[][] a) throws IOException`

Writes a two-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

TextDataWriter

Text data writer. Writes fields as columns or as rows in a text file.

```
package umontreal.iro.lecuyer.util.io;

public class TextDataWriter extends CachedDataWriter
```

Fields

```
public final String DEFAULT_COLUMN_SEPARATOR = "";
```

Default value for the column separator.

```
public final String DEFAULT_HEADER_PREFIX = "";
```

Default value for the header prefix.

Enums

```
public enum Format { COLUMNS, ROWS }
```

Output format: organize fields as columns or as rows.

Constructors

```
public TextDataWriter (String filename, Format format, boolean withHeaders)
      throws IOException
```

Class constructor. Truncates any existing file with the specified name.

```
public TextDataWriter (File file, Format format, boolean withHeaders)
      throws IOException
```

Class constructor. Truncates any conflicting file.

```
public TextDataWriter (OutputStream outputStream, Format format,
                       boolean withHeaders)
      throws IOException
```

Class constructor.

Methods

`public void setFormat (Format format)`

Changes the output format.

`public void setFloatFormatString (String formatString)`

Sets the format string used to output floating point numbers.

`public void setColumnSeparator (String columnSeparator)`

Changes the column separator.

`public void setHeaderPrefix (String headerPrefix)`

Changes the header prefix (a string that indicates the beginning of the header line for the `COLUMNS` format).

`public void close() throws IOException`

Flushes any pending data and closes the file or stream.

BinaryDataWriter

Binary data writer.

Stores a sequence of fields in binary file, which can be either atoms or arrays, each of which having the following format:

- Field label:
  - Pipe character (`|`)
  - Label length (32-bit integer, big endian)
  - Label string (array of bytes of the specified length)
- Field type (byte):
  - `i` (32-bit integer)
  - `f` (32-bit float)
  - `d` (64-bit double)
  - `S` (string)
- Number of dimensions (8-bit integer)
- Dimensions (array of 32-bit integers, big endian)
- Field data (in the specified format, big endian)

In the case of an atomic field, the number of dimensions is set to zero.

A string field is stored in the following format:

- String length (32-bit integer)
- Array of bytes of the specified length

Also supports anonymous fields (fields with an empty label).

Arrays up to two dimensions are supported.

Modules for reading data exported with this class are available in Java (`BinaryDataReader`), Matlab and Python (numpy).

```
package umontreal.iro.lecuyer.util.io;

public class BinaryDataWriter extends AbstractDataWriter
```

Fields

`public final static byte TYPECHAR_LABEL = '|';`
Field-type symbol indicating a label (it more accurately a field separator symbol).

`public final static byte TYPECHAR_STRING = 'S';`
Field-type symbol indicating `String` data.

`public final static byte TYPECHAR_INTEGER = 'i';`
Field-type symbol indicating `int` data.

`public final static byte TYPECHAR_FLOAT = 'f';`
Field-type symbol indicating `float` data.

`public final static byte TYPECHAR_DOUBLE = 'd';`
Field-type symbol indicating `double` data.

Constructors

```
public BinaryDataWriter (String filename, boolean append)
      throws IOException
```
Data will be output to the file with the specified name.

`public BinaryDataWriter (File file, boolean append) throws IOException`
Data will be output to the specified file.

`public BinaryDataWriter (String filename) throws IOException`
Truncates any existing file with the specified name.

`public BinaryDataWriter (File file) throws IOException`
Truncates any existing file with the specified name.

`public BinaryDataWriter (OutputStream outputStream) throws IOException`
Constructor.

Writing atomic data

`public void write (String label, String s) throws IOException`
Writes an atomic string field. Writes an anonymous field if `label` is `null`.

`public void write (String label, int a) throws IOException`
Writes an atomic 32-bit integer (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float a) throws IOException`
Writes an atomic 32-bit float (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double a) throws IOException`
Writes an atomic 64-bit double (big endian). Writes an anonymous field if `label` is `null`.

Writing one-dimensional arrays

`public void write (String label, String[] a, int n) throws IOException`

Writes the first `n` elements of a one-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, int[] a, int n) throws IOException`

Writes the first `n` elements of a one-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[] a, int n) throws IOException`

Writes the first `n` elements of a one-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[] a, int n) throws IOException`

Writes the first `n` elements of a one-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

Writing two-dimensional arrays

`public void write (String label, String[][] a) throws IOException`

Writes a two-dimensional array of strings. Writes an anonymous field if `label` is `null`.

`public void write (String label, int[][] a) throws IOException`

Writes a two-dimensional array of 32-bit integers (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, float[][] a) throws IOException`

Writes a two-dimensional array of 32-bit floats (big endian). Writes an anonymous field if `label` is `null`.

`public void write (String label, double[][] a) throws IOException`

Writes a two-dimensional array of 64-bit doubles (big endian). Writes an anonymous field if `label` is `null`.

Other methods

`public void close() throws IOException`

Flushes any pending data and closes the file.

DataReader

Data reader interface.

```
package umontreal.iro.lecuyer.util.io;

public interface DataReader
```

Reading atomic data

`public String readString (String label) throws IOException;`

Reads the first field labeled as `label` and returns its `String` value.

`public int readInt (String label) throws IOException;`

Reads the first field labeled as `label` and returns its `int` value.

`public float readFloat (String label) throws IOException;`

Reads the first field labeled as `label` and returns its `float` value.

`public double readDouble (String label) throws IOException;`

Reads the first field labeled as `label` and returns its `double` value.

Reading one-dimensional arrays

`public String[] readStringArray (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a one-dimensional array of `String`'s.

`public int[] readIntArray (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a one-dimensional array of `int`'s.

`public float[] readFloatArray (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a one-dimensional array of `float`'s.

`public double[] readDoubleArray (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a one-dimensional array of `double`'s.

Reading two-dimensional arrays

`public String[][] readStringArray2D (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a two-dimensional array of `String`'s.

`public int[][] readIntArray2D (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a two-dimensional array of `int`'s.

`public float[][] readFloatArray2D (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a two-dimensional array of `float`'s.

`public double[][] readDoubleArray2D (String label) throws IOException;`

Reads the first field labeled as `label` and returns its value as a two-dimensional array of `double`'s.

Reading fields of unknown type

`public Map<String, DataField> readAllNextFields() throws IOException;`

Reads all remaining fields in the file and returns a hashmap indexed by field labels. Anonymous fields are mapped to `"_data01_"`, `"_data02_"`, ...

`public Map<String, DataField> readAllFields() throws IOException;`

Reads all fields in the file and returns a hashmap indexed by field labels. Anonymous fields are mapped to `"_data01_"`, `"_data02_"`, ...

`public DataField readNextField() throws IOException;`

Reads the next available field.

`public DataField readField (String label) throws IOException;`

Reads the first field labeled as `label`.

Other methods

`public void close() throws IOException;`

Closes the input stream.

`public void reset() throws IOException;`

Resets the reader to its initial state, i.e. goes back to the beginning of the data stream, if possible.

`public boolean dataPending() throws IOException;`

Returns `true` if there remains data to be read.

AbstractDataReader

This abstract class implements shared functionality for data readers.

```
package umontreal.iro.lecuyer.util.io;

public abstract class AbstractDataReader implements DataReader
```

Reading atomic data

`public String readString (String label) throws IOException`
Reads first field labeled as `label` and returns its `String` value.

`public int readInt (String label) throws IOException`
Reads first field labeled as `label` and returns its `int` value.

`public float readFloat (String label) throws IOException`
Reads first field labeled as `label` and returns its `float` value.

`public double readDouble (String label) throws IOException`
Reads first field labeled as `label` and returns its `double` value.

Reading one-dimensional arrays

`public String[] readStringArray (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a one-dimensional array of `String`'s.

`public int[] readIntArray (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a one-dimensional array of `int`'s.

`public float[] readFloatArray (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a one-dimensional array of `float`'s.

`public double[] readDoubleArray (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a one-dimensional array of `double`'s.

Reading two-dimensional arrays

`public String[][] readStringArray2D (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a two-dimensional array of `String`'s.

`public int[][] readIntArray2D (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a two-dimensional array of `int`'s.

`public float[][] readFloatArray2D (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a two-dimensional array of `float`'s.

`public double[][] readDoubleArray2D (String label) throws IOException`
Reads first field labeled as `label` and returns its value as a two-dimensional array of `double`'s.

Reading fields of unknown type

`public Map<String, DataField> readAllNextFields() throws IOException`

Reads all remaining fields in the file and returns a hashmap indexed by field labels. Anonymous fields are mapped to `"_data01_"`, `"_data02_"`, ...

`public Map<String, DataField> readAllFields() throws IOException`

Reads all fields in the file and returns a hashmap indexed by field labels. Anonymous fields are mapped to `"_data01_"`, `"_data02_"`, ...

BinaryDataReader

Binary data reader. This class implements a module for importing data written with `BinaryDataWriter`.

```
package umontreal.iro.lecuyer.util.io;

public class BinaryDataReader extends AbstractDataReader
```

Constructors

`public BinaryDataReader (String filename) throws IOException`

Opens the file with the specified name for reading.

`public BinaryDataReader (URL url) throws IOException`

Opens the file at the specified url for reading.

`public BinaryDataReader (File file) throws IOException`

Opens the specified file for reading.

`public BinaryDataReader (InputStream inputStream) throws IOException`

Opens the specified input stream for reading. When using this constructor, the method `readField` might will not be able to read a field that is before the current reading position.

Reading fields of unknown type

`public DataField readNextField() throws IOException`

Reads the next available field.

`public DataField readField (String label) throws IOException`

Reads the first field labeled as `label`.

Other methods

`public void reset() throws IOException`

Reopens the file (does not work with the constructor that takes an input stream).

`public boolean dataPending() throws IOException`

Returns `true` if there remains data to be read.

`public void close() throws IOException`

Closes the file.

DataField

This class represents a data field from a file read by an instance of a class implementing `DataReader`.

```
package umontreal.iro.lecuyer.util.io;

public class DataField
```

Constructors

`public DataField (String label, Object data)`
Constructor. Creates a field named `label` of value `data`.

`public DataField (String label, Object data, int effectiveLength)`
Constructor. Creates a field named `label` of value `data`. `effectiveLength` is the number of significant elements contained in `data` if it is an array.

Information on the field

`public String getLabel()`
Returns the field label (or name).

`public Class getType()`
Returns the type of the field.

`public boolean isAtomic()`
Returns `true` if the field value is atomic data.

`public boolean isArray()`
Returns `true` if the field contains an array.

`public boolean isArray2D()`
Returns `true` if the field contains a two-dimensional array.

`public int getArrayLength()`
Returns the length of the array contained by the field, or `-1` if it is not an array.

`public boolean isString()`
Returns `true` if the field value is an atomic `String`.

`public boolean isInt()`
Returns `true` if the field value is an atomic `int`.

`public boolean isFloat()`
Returns `true` if the field value is an atomic `float`.

`public boolean isDouble()`
Returns `true` if the field value is an atomic `double`.

Obtaining the value as atomic data

`public String asString()`
Returns the value as `String`, or `null` if it is not of type `String`. See `isString`.

`public int asInt()`
Returns the value as `int` or `0` if it is not of type `int` See `isInt`.

`public float asFloat()`
Returns the value as `float` or `0` if it is not of type `float` See `isFloat`.

`public double asDouble()`
Returns the value as `double` or `0` if it is not of type `double` See `isDouble`.

Obtaining the value as a one-dimensional array

`public String[] asStringArray()`
Returns the value as one-dimensional `String` array or `null` if it is not of type `String[]`.

`public int[] asIntArray()`
Returns the value as one-dimensional `int` array or `null` if it is not of type `int[]`.

`public float[] asFloatArray()`
Returns the value as one-dimensional `float` array or `null` if it is not of type `float[]`.

`public double[] asDoubleArray()`
Returns the value as one-dimensional `double` array or `null` if it is not of type `double[]`.

Obtaining the value as a two-dimensional array

`public String[][] asStringArray2D()`
Returns the value as two-dimensional `String` array or `null` if it is not of type `String[][]`.

`public int[][] asIntArray2D()`
Returns the value as two-dimensional `int` array or `null` if it is not of type `int[][]`.

`public float[][] asFloatArray2D()`
Returns the value as two-dimensional `float` array or `null` if it is not of type `float[][]`.

`public double[][] asDoubleArray2D()`
Returns the value as two-dimensional `double` array or `null` if it is not of type `double[][]`.

Obtaining the value as an `Object`

`public Object asObject()`
Returns the value of the field as an `Object`.

References

[1] C. W. Clenshaw. Chebychev series for mathematical functions. National Physical Laboratory Mathematical Tables 5, Her Majesty's Stationery Office, London, 1962.

[2] J. Gosling, B. Joy, and G. L. Steele Jr. *The Java Language Specification*. Addison-Wesley, second edition, 2000. Also available from `http://java.sun.com/docs/books/jls`.

[3] N. J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM Review*, 51(4):747–764, 2009.

[4] D. E. Knuth. *The Art of Computer Programming, Vol. 1*. Addison-Wesley, Reading, MA, second edition, 1973.

[5] B. Skaflestad and W. M. Wright. The scaling and modified squaring method for matrix functions related to the exponential. *Applied Numerical Mathematics*, 59(3-4):783–799, 2009.